Just-Right Consistency: reconciling availability and safety

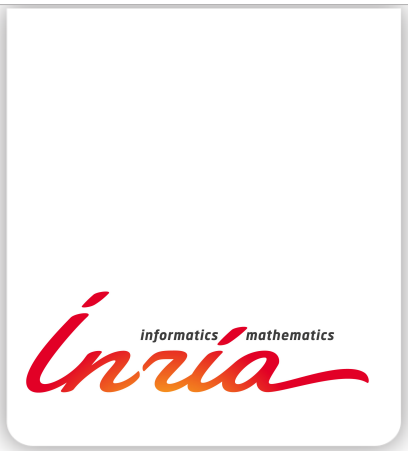
Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, Christopher Meiklejohn

Just-Right Consistency: reconciling availability and safety

Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, Christopher Meiklejohn

RESEARCH REPORT
N° 9145
January 2018
Project-Team Delys

ISSN 0249-6399 ISRN INRIA/RR--9145--FR+ENG



Just-Right Consistency: reconciling availability and safety

Marc Shapiro*, Annette Bieniusa†, Nuno Preguiça‡, Valter Balegas§, Christopher Meiklejohn¶

Project-Team Delys

Research Report n° 9145 — January 2018 — 14 pages

* Sorbonne-Université—LIP6—Inria Paris
† T. U. Kaiserslautern
‡ U. Nova de Lisboa
§ U. Nova de Lisboa
¶ U. Catholique de Louvain

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Abstract: By the CAP Theorem, a distributed data storage system can ensure either Consistency under Partition (CP) or Availability under Partition (AP), but not both. This has led to a split between CP databases, in which updates are synchronous, and AP databases, where they are asynchronous. However, there is no inherent reason to treat all updates identically: simply, the system should be as available as possible, and synchronised just enough for the application to be correct. We offer a principled *Just-Right Consistency* approach to designing such applications, reconciling correctness with availability and performance, based on the following insights: *(i)* The Conflict-Free Replicated Data Type (CRDTs) data model supports asynchronous updates in an intuitive and principled way. *(ii)* Invariants involving joint or mutually-ordered updates are compatible with AP and can be guaranteed by Transactional Causal Consistency, the strongest consistency model that does not compromise availability. Regarding the remaining, "CAP-sensitive" invariants: *(iii)* For the common pattern of Bounded Counters, we provide encapsulated data type that is proven correct and is efficient; and *(iv)* in the general case, static analysis can identify when synchronisation is not necessary for correctness. Our Antidote cloud database system supports CRDTs, Transactional Causal Consistency and the Bounded Counter data type. Support tools help design applications by static analysis and proof of CAP-sensitive invariants. This system supports industrial-grade applications and has been tested experimentally with hundreds of servers across several geo-distributed data centres.

Key-words: Distributed systems; distributed programming; consistency; availability; invariants; CAP Theorem

La juste cohérence pour reconcilier disponibilité et sûreté

Résumé : Le théorème CAP, un système de stockage réparti peut être, en cas de partition, soit cohérent (CP), soit disponible (AP), mais pas les deux. Il y a donc des bases de données CP, où les mises à jour sont synchrones, et les bases AP, où elles sont asynchrones. Cependant, il n'y a pas de raison essentielle de traiter toutes les mises à jour de façon identique. L'objectif est que le système reste aussi disponible que possible, mais suffisamment synchronisé pour que l'application reste correcte. Nous proposons un nouveau principe, la *juste cohérence*, afin de concevoir de telles applications, réconciliant la sûreté avec la disponibilité et l'efficacité, à partir des constatations suivantes : *(i)* Le modèle de données des CRDT (Conflict-Free Replicated Data Type) permet les mises à jour concurrentes de façon à la fois théoriquement fondée et intuitive. *(ii)* Les invariants basés sur la simultanéité ou l'ordre partiel des mises à jour sont compatibles avec AP, et peuvent être garanties par la Cohérence Causale Transactionelle (TCC), le modèle de cohérence le plus fort qui ne compromet pas la disponibilité.

En ce qui concerne les autres invariants, dits *CAP-sensibles* : *(iii)* Le cas courant du compteur borné peut être géré par un type de données encapsulé, correct et cohérent, appelé *Bounded Counter* ; et *(iv)* dans le cas général, une analyse statique permet d'indentifier les cas où la sûreté ne nécessite pas de synchronisation. Notre base de données "nuage" Antidote offre les CRDT, le modèle TCC, et le type de données Bounded Counter. Des outils d'analyse statique et de preuve des invariants CAP-sensibles aident à la conception des applications. Notre système est mûr pour des applications d'échelle industrielle, et a été testé expérimentallement sur des centaines de serveurs répartis entre plusieurs centres de données géo-distribués.

Mots-clés : Système distribué ; programmation répartie ; cohérence ; disponibilité ; invariants ; théorème CAP

Contents

1 Introduction: The CAP gap

Many modern applications store their data in a *cloud database system* in order to distribute and replicate data over many servers, across geo-distributed data centres (DCs). Application developers are faced with unfamiliar behaviours and unpredictability, bringing complexity and new opportunities for error.

Therefore, some database systems provide "strong consistency," which mimics a sequential, centralised system. Examples include the Spanner/F1 system [11] or the Serialisability model.[1] Under the hood, operations synchronise to maintain the illusion of a total order. If the network is *partitioned*, synchronisation blocks and the database waits indefinitely (until the partition is repaired): the system remains *Consistent under Partition (CP)*, but is not available. As synchronisation between geographically remote DCs waits for tens or hundreds of milliseconds, this has a performance cost. Thus, Spanner/F1 requires around 100 ms to commit an update transaction [11]. CP is overly conservative for many applications.

Alternatively, the system might access the local replica without synchronising. Latency is minimal, transactions run in parallel, and the system is *Available under Partition (AP)* [1]. However, a read might return a stale value and writes may conflict. In this vein, early AP systems, such as Cassandra [16], Dynamo [12] or Riak [10] implement Eventual Consistency, which provides very weak guarantees.

The CAP Theorem states that a system cannot be both CP and AP [14]. It seems there are only two alternatives: a conservative CP system that makes guarantees, but has high cost and low availability; or a bold AP system that is efficient and available, but has correctness problems.

This is a false dichotomy. Our insight is to *tailor consistency to application requirements*, and not shoe-horn applications into a rigidly-defined consistency model. We consider the system is correct if it maintains application-level *integrity invariants*. Different applications (or even parts thereof) have very different invariants. For instance, many social networks work fine above an AP database. In contrast, a banking application would seem to require CP to maintain the "no overdraft" invariant. Notice however that this application is still correct if deposit operations are non-synchronised (i.e., AP) [18]; withdrawals themselves must synchronise only when the balance is low [8]!

In the rest of this paper, we build upon this intuition to develop the *Just-Right Consistency* approach. Our aims is to make the application as available as possible, but synchronised enough to remain correct. Our base model, Transactional Causal Consistency, maintains *AP-compatible* invariant patterns, and we switch to CP selectively when provably required for a *CAP-sensitive* invariant.

2 Keep my app safe!

To be concrete, let's consider an example, the FMKe application [26]. FMKe is modelled after the Danish National Joint Medicine Card system FMK (*Fælles Medicinkort*), which concerns every Danish citizen and has been running 24×7 since 2013 [26]. FMKe handles the lifecycle of prescriptions and events associated with patients, doctors, hospitals and pharmacies. Its major

[1] Serialisability is characterised by the ACID properties: All-or-Nothing, Correct-Individually, Isolation or total order, and Durability. The first three are defined later in this paper. Durability means that the result of some update is observed by all later reads, for some definition of "later."

operations are the following (each application-level operation comprises a number of database reads and/or updates):

- create-prescription creates a prescription record associated with a patient, a doctor, and a pharmacy.
- update-prescription-medication adds a medication to a prescription, or increases the count associated with a medication.
- process-prescription corresponds to delivering medication by a pharmacy.
- get-staff-prescriptions and get-pharmacy-prescriptions return the prescriptions associated with a given staff member and pharmacy respectively.

Let's first consider how this application maintains its invariants in a sequential setting, in order to identify important patterns.

Relative-order pattern Our first programming pattern for preserving invariants leverages the relative order of database accesses. For instance, in FMKe, create-prescription first initialises a prescription record, then makes the relevant patient record point to it. Changing this order would violate the "referential integrity" invariant.

Joint-update pattern Our second pattern concerns joint updates to separate data items. FMKe offers several examples. For instance, creating a prescription updates not just the patient record but also the corresponding doctor and pharmacy records. When the underlying database is non-normalised, FMKe maintains separate but identical copies of the prescription in each of these records. In both cases, any other operation accessing the database must observe the state, either before the joint update takes place (there was no change), or after (all the joint changes took effect), and will never observe an intermediate state.

Precondition-check pattern The final pattern is conditioning an update to a *precondition check*. For instance, a prescribed medication comes with a count of how many times it can be delivered to the patient. To deliver one box of the medication, process-prescription checks precondition count ≥ 1, and, if true, decrements count by one.

The Correct-Individually assumption Even when the developer does not think explicitly in terms of invariants, she uses the above patterns, individually or in combination, to maintain invariants implicitly. Informally, relative-order updates maintain a partial order between data items; joint-updates maintain equivalence between different instances of the same information; and precondition checks serve to maintain value-based assertions [24]. The developer must be careful to apply these patterns to maintain the underlying invariants, even in sequential code.[2]

Thus, we can make the critical assumption that the application "does the right thing" in a sequential execution (if it is incorrect sequentially, then discussion of consistency is moot). Technically, we require that, if the invariants are true in some state of the database and an operation executes, in the state after the operation the invariants remain true. We say that each operation is *correct individually* (the "C" in ACID).

[2] We argue that these three patterns are the critical ones. Indirect evidence for this conjecture is that the strongest consistency models preserve these patterns, and transparently guarantee application invariants [24].

3 AP-compatible invariant patterns

Reasoning about asynchronous updates is difficult. This section first presents the CRDT data model; then we discuss an AP consistency model that preserves the relative-order and joint-update invariant patterns.

3.1 Data model: CRDTs

Because concurrent assignments do not commute, they may not be concurrent: ordinary assignments require the CP sledgehammer. AP data needs an alternative data model suitable for concurrent updates.

A common approach (e.g., in Cassandra) is "Last-Writer-Wins," where concurrent assignments to the same location are resolved in favour of the one with the highest timestamp; the other one is a "lost update." A higher-level approach uses Conflict-free Replicated Data Types (CRDTs) [23]. A CRDT extends a sequential abstract data type, and ensures by construction that concurrent updates are merged deterministically and replicas converge. There are CRDTs for many familiar abstractions, including registers, counters, sets, maps, graphs and sequences. For instance, two doctors could concurrently add elements Aspirin and Chamomile to a CRDT set object; as expected, both elements will be in the set. Non-commuting updates are resolved according to application requirements; for instance, when concurrently adding and removing the same element, add might win. Due to space constraints, we refer the reader interested in knowing more about CRDTs to the literature.

3.2 The relative-order pattern: Causal Consistency

Remember how ensuring the referential integrity invariant relies on the order in which operations occur. More generally, applications often use ordering to ensure an *implication* invariant $P \implies Q$, by first making Q true, then P.[3] If some replica observed the updates in a different order, the invariant could be violated.

Here is an example. Database BuggyDB comes with a default admin password of 0000 and admin login disabled. Cindy, working from the Copenhagen replica, sets the password to S3kr3t, then enables login. BuggyDB does not guarantee to make these updates visible in the same order. Malicious user Moriarty, who is accessing a replica in Middelfart (centre of Denmark), notices admin login is enabled and gains control with the default password 0000. The (implicit) invariant "admin login enabled $\implies$ password $\neq$ default" has been violated.

To transparently guarantee the relative-order pattern, the system should ensure that related updates become visible in the same order at all replicas. A common approach, called Causal Consistency (CC) [2], is based on Lamport's happened-before relation [17]: *(i)* If an application thread performs update u followed by update v, or *(ii)* if the application reads from update u and later performs update v, or *(iii)* any transitive combination of the above, then a CC database makes u visible before v. Unrelated (concurrent) updates can become visible in any order. CC does not impact availability, because the database can always read some version, and concurrent CRDT writes are merged.

If we assume that the application performs its updates in the right order (which it must do, since otherwise even sequential execution would be incorrect), Causal Consistency guarantees that

[3] This approach is derived from the application-level "demarcation protocol" [9].

the corresponding invariants will be maintained transparently. No extra work is required from the developer; in particular, she does not need to explicitly understand or write out the invariants.

3.3 Joint-update pattern: AP transactions

A joint update is a limited form of transaction [5]. It requires the *All-or-Nothing* property (the "A" in ACID). An FMKe example is creating a prescription, which jointly updates the corresponding patient, doctor and pharmacy records.

One part of All-or-Nothing is to ensure that, at every replica, either all the updates of a transaction are visible together, or none is at all.

An incorrectly designed system might violate this property. Example: Dr. Alice in Aalborg (a city in the North of Denmark) adds Aspirin to Bob's prescription. This updates both Bob's patient copy and the pharmacy copy. The underlying BuggyDB2 database in Aalborg pushes its patient-record updates to the replica in Byrum (on the Læsø island), but not its pharmacy-record updates (perhaps they are assigned to different servers). Bob's local pharmacy in Byrum observes that Aspirin appears in Bob's prescription but, incorrectly, not in the pharmacy's copy.

Clearly, the system should instead transport all the updates of a transaction as a single unit, even if they belong to different servers (this property is called atomic writes). This does not impact availability: if Byrum is partitioned from Aalborg, Byrum sees no change; after communication is restored, Byrum sees both updates. Either way, both sites remain available to their local users.

The complementary *snapshot property* is often overlooked. It states that all of a transaction's reads must come from (updates by) the same set of transactions, called its snapshot, even if reads are served by different replicas.

Let's consider the previous example above BuggyDB3, which implements atomic writes but not snapshots. Suppose that Dr. Alice created Bob's prescription in Transaction T1, then added Aspirin in Transaction T2. Transaction T3 reads the patient record written by T1 and the pharmacy record written by T2. It will find Aspirin in the latter's copy of the prescription but not in the former, violating their equality.

All-or-Nothing (the conjunction of atomic writes and snapshots) avoids such "broken reads;" snapshots are also instrumental in ensuring that transactions satisfy Causal Consistency. Let's say T3's snapshot contains T1 and T2. Then T3 would observe the prescription set by T1, and both prescription updates of T2.

If the application developer carefully groups its operations into transactions (a small price to pay), a database that ensures the All-or-Nothing properties will transparently guarantee the corresponding invariants. No extra work is required from the developer; in particular, she does not need to explicitly understand or write out the invariants.

3.4 Transactional Causal Consistency, the strongest AP model

Transactions and Causal Consistency are found in many CP models, such as Strict Serialisability or Snapshot Isolation. However, many database models (even CP ones, such as Serialisability) do not enforce condition *(i)* of Causal Consistency, and therefore could fail the password example.

First-generation AP systems, such as Cassandra [16] or Riak [10], do not support transactions nor Causal Consistency. This is unfortunate, because these mechanisms are compatible with AP.

Without them, developers have a hard time to reason about the behaviour of their application, as some basic expectations are violated. Many applications have implicit invariants that require proper ordering and grouping [6].

The AP model that enforces Causal Consistency and All-or-Nothing is called Transactional Causal Consistency (TCC). TCC is the strongest model that does not compromise availability.[4] Recent research systems such as COPS [19], Eiger [20], GentleRain [13] or SwiftCloud [27] provide restricted variants of TCC. The open-source, CRDT-based database Antidote, which we developed in the SyncFree European Project, is the first industrial-strength (now in alpha) geo-replicated database system with a fully functional and unrestricted implementation of TCC [3, 25].

4 CAP-sensitive invariant patterns

Finally, we discuss invariants that are *not* AP-compatible. Before we discuss the general case, in Section 4.2, let's first consider how to address a restricted but useful case.

4.1 A specific case: Bounded Counter data type

A common case of CAP-sensitive problem is maintaining a shared counter x, which supports increment and decrement operations but must remain above some parameter k. By applying *escrow* techniques [22], carefully caching partial state, batching synchronisation, and moving communication off the critical path, the counter can maintain the invariant $x \geq k$, while remaining efficient and mostly-AP.

We implement this algorithm in a specialised data type, the Bounded Counter [8]. Skipping the technical details, we illustrate Bounded Counter with an example. Consider maintaining the budget of the health system with a Bounded Counter constrained to remain non-negative ($k = 0$). It may be incremented (e.g., receiving payments) or decremented (e.g., purchasing inventory). Clearly, increments cannot violate the invariant; therefore increment can run in AP mode. Furthermore, a pharmacy might donate some of its available share to another one, even before it is needed; donate is also an AP operation.[5] This is in contrast to decrement; however, instead of synchronising every decrement, we can pre-allocate some share of the budget to each pharmacy and hospital. As long as the local share remains sufficient, decrement affects only the local share, in AP mode. It is only if the local share is too small that decrement must synchronise. Note that decrement risks unavailability only in this rare case.

We encapsulate the implementation within the Bounded Counter provided with Antidote. The developer does not need to understand the details; she just needs to set the bound, initial value and initial per-replica share. Then the application calls increment, decrement, and possibly donate, as appropriate for the application; synchronisation remains transparent. The Bounded Counter algorithm has been proven correct using the general techniques of Section 4.3.

[4] Attiya et al. [4] call Causal Consistency the strongest AP model, but they do not consider transactions, only single operations.

[5] Operation donate uses the demarcation protocol [9], which requires Causal Consistency.

4.2 The problem with precondition checks

Let's now (finally!) consider the general case of the precondition-check pattern, of which Bounded Counter is just a restricted example. Unfortunately, this pattern is *CAP-sensitive*, because checking the local replica might be unsafe in an AP system: even if two replicas have the same state, one might test the precondition to be true, while concurrently the other replica is making an update that causes it to become false; when the second update gets delivered to the first replica, the invariant is violated. We say the precondition is not *stable under concurrent update* [15].

In FMKe, process-prescription checks that count ≥ 1 and decrements count, in order to avoid that medication is delivered in duplicate. Now, let's say Bob in Byrum has a prescription for one box of Aspirin. Bob, and his accomplice Moriarty in Middelfart, present this same prescription to their local pharmacies at the same time. At both replicas, the precondition count ≥ 1 holds; inherently to AP, a pharmacy is unaware of the other's concurrent actions; both decrement count and deliver the medication, incorrectly. The reason is that the precondition evaluates to true at the first replica, but is negated by the concurrent execution of process-prescription at another.

The only way to be sure the invariant will not be violated is to prohibit this concurrency (i.e., to synchronise). Must we admit defeat, and adopt the CP model and impose a total order over all operations (the I for Isolation property of ACID)? No, this would be overkill. Different operations have different requirements, and even for a CAP-sensitive invariant, not all executions need to be synchronised.

For instance, since the get-* operations are read-only, they do not change the truth of the process-prescription precondition. Furthermore, even though update-prescription-medication changes the count of a medication, it only increases it, which cannot negate the precondition. In other words, the precondition of process-prescription is stable under concurrent get-* or update-prescription-medication, and it is safe to let them run concurrently.

When a precondition is unstable, the developer has exactly two alternatives: *(i)* Either to forbid concurrency, in order to avoid negating the precondition check; the update runs in CP mode, at the expense of availability; or *(ii)* to remain available, but accept that the invariant might be violated (in which case it is not a real invariant!). If the FMKe developer chooses the first option, she instructs the database to forbid two process-prescription operations concerning the same prescription from running concurrently; then, a user will not be able to get her medication when the network is partitioned. The second option is to downgrade the invariant to a best-effort objective, or even to remove the check altogether, and risk delivering a medication in duplicate. This is a design decision: it's a trade-off between the availability *of this particular operation* and the value *of this particular invariant.* In fact, for the designers of the FMK production system, availability was the top design objective, and they chose the second option, accepting a non-zero probability of delivering medication in duplicate.

If the developer wishes to make the opposite decision, and enforce a CAP-sensitive invariant, what are her choices? To remain as available as possible, we wish to synchronise *only when strictly necessary.* In this example, we would forbid running two concurrent process-prescription relating to the same prescription, but allow it for different prescriptions. We would also let process-prescription run concurrently with get-* or update-prescription-medication.

4.3 Verifying general CAP-sensitive invariants

We now understand why certain updates need to synchronise, and others not. But this is getting complicated. How is a developer to get it right? With too little synchronisation, invariants can

be violated; with too much, availability and performance suffer. Bailis et al. [6] show that an ad-hoc approach is error prone.

The bad news is that, outside of the strongest CP models, avoiding unstable invariants is not transparent and requires knowledge of the application. The good news is that we have developed tools to automate this analysis, and ensure that there are no mistakes — to verify statically, i.e., at design time, that your invariants are verified, even though most operations remain available. No guesswork!

Consider our CISE tool [15, 21]. Given the application specification (expressed in first-order logic), CISE checks the following conditions: *(i)* operations are Correct Individually (see Section 2); *(ii)* concurrent updates converge (see Section 3.1), and *(iii)* every precondition is stable with respect to concurrent updates. If all three checks pass, this constitutes a formal proof that the application invariant remains true at all times when the application runs above a Causally Consistent database [15]. Otherwise, the tool returns a counter-example, which the developer can use to diagnose the cause of the problem. The fix can either be to change the application semantics in order to remain AP, or to synchronise the two updates (switching to CP, but only when strictly necessary).

Checking the FMKe application runs like this. The invariant to verify is that a medication is not delivered more times than prescribed. First, the tool verifies that, for every FMKe operation in isolation, with any legal parameter value, its precondition implies the invariant. This check passes, because update-prescription-medication can only increase count and because process-prescription checks the remaining count of every medication, and decreases that count by what is delivered.

Second, it verifies that replicas will converge, by comparing that all pairs of concurrent operations (with any parameter), yield the the same database state when run in opposite orders. This check passes, for the following reasons. The get-* operations have no side effects, therefore they commute with all operations. The create-prescription is necessarily causally before any other operation on the same prescription, hence not concurrent with it. The update-prescription-medication and process-prescription operations operate on CRDTs, which converge by construction.

For precondition stability, the tool checks that no update, with any argument, ever negates the precondition check of a concurrent update. This check fails, returning the following counter-example. It starts with a prescription containing a medication count of one, and performing process-prescription twice concurrently. The precondition check is not stable since one tests count to be 1, and the other changes it to 0. This shows that, to maintain the invariant, process-prescription must synchronise with other process-prescriptions of the same prescription. If we add this synchronisation to the application, we can run the tool again; this time the verification succeeds. Alternatively (following the FMK design explained in Section 4), we can remove the "no duplicates" invariant; this also causes verification to succeed.

In order to support the CISE analysis, Antidote will run specific transactions in a CP mode that upholds both TCC and the ACID properties.

5 Conclusion

Developing correct and highly-scalable applications is a challenging task. Instead of shoe-horning an application to a rigidly-defined consistency model, we advocate a new Just-Right Consistency approach, focusing on *maintaining the application invariants* that are already present in a sequential environment.

Based on an appropriate data model, CRDTs, we showed which invariant patterns are AP-compatible, and how they can be guaranteed transparently in an AP system. Accordingly, we recommend Transactional Causal Consistency as the default consistency model. Our Antidote open-source, CRDT-based database is the first one to fully implement TCC.

The remaining patterns are sensitive to the CAP gap between safety and availability. The Bounded Counter (one of the data types supported by Antidote) constitutes a pre-packaged solution to a common case, encapsulating the necessary synchronisation and minimising its impact. For the general CAP-sensitive case, the CISE logic and tools verifies whether an application has sufficient synchronisation, and if not, helps identify the offending operations. This enables tailoring synchronisation precisely to the application requirements.

Acknowldegements

The Just-Right Consistency concept derives from previous work by Balegas et al. [7] and has benefited from discussion with Masoud Saieda Ardekani and Alexey Gotsman. The Bounded Counter concept is due to Balegas et al. [8]. The CISE logic is due to Gotsman et al. [15]; the CISE tool was conceived and implemented by Mahsa Najafzadeh [21]. FMKe was designed based on discussion with Kresten Krab Thorup.

Thanks to the whole Antidote team, who made this work possible: Deepthi Akkoorath, Valter Balegas, Manuel Bravo, Tyler Crain, Viktória Fördős, Michał Jabczyński, Zhongmiao Li, Ali Shoker, Gonçalo Tomás, Alejandro Tomsic, and Peter Zeller.

This research is supported in part by European projects SyncFree (FP7 609 551), and LightKone (H2020 732 505)

References

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, Feb. 2012.

[2] M. Ahamad, G. Neiger, J. E. Burns, et al. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar. 1995.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, et al. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pp. 405–414, Nara, Japan, June 2016.

[4] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 28(1):141–155, Jan. 2017.

[5] P. Bailis, A. Davidson, A. Fekete, et al. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, Nov. 2013.

[6] P. Bailis, A. Fekete, M. J. Franklin, et al. Feral concurrency control: An empirical investigation of modern application integrity. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pp. 1327–1342, Melbourne, Victoria, Australia, 2015.

[7] V. Balegas, N. Preguiça, R. Rodrigues, et al. Putting consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pp. 6:1–6:16, Bordeaux, France, Apr. 2015.

[8] V. Balegas, D. Serra, S. Duarte, et al. Extending eventually consistent cloud databases for enforcing numeric invariants. In *Symp. on Reliable Dist. Sys. (SRDS)*, pp. 31–36, Montréal, Canada, Sept. 2015. Not open access.

[9] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal, The Int. J. on Very Large Data Bases*, 3(3):325–353, July 1994.

[10] Basho, Inc. Riak KV: Distributed NoSQL database. Website http://basho.com/products/riak-kv/, 2016. Accessed 4 June 2016.

[11] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 251–264, Hollywood, CA, USA, Oct. 2012.

[12] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pp. 205–220, Stevenson, Washington, USA, Oct. 2007.

[13] J. Du, C. Iorgulescu, A. Roy, et al. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing*, pp. 4:1–4:13, Seattle, WA, USA, Nov. 2014.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[15] A. Gotsman, H. Yang, C. Ferreira, et al. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 371–384, St. Petersburg, FL, USA, 2016.

[16] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, Apr. 2010. W. on Large-Scale Dist. Sys. and Middleware (LADIS) 2009.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] C. Li, D. Porto, A. Clement, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 265–278, Hollywood, CA, USA, Oct. 2012.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 401–416, Cascais, Portugal, Oct. 2011.

[20] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pp. 313–328, Lombard, IL, USA, Apr. 2013.

[21] M. Najafzadeh, A. Gotsman, H. Yang, et al. The CISE tool: Proving weakly-consistent applications correct. In *W. on Principles and Practice of Consistency for Distr. Data (PaPoC)*, EuroSys 2016 workshops, London, UK, Apr. 2016.

[22] P. E. O'Neil. The escrow transactional method. *Trans. on Database Systems*, 11(4):405–430, Dec. 1986. ISSN 0362-5915.

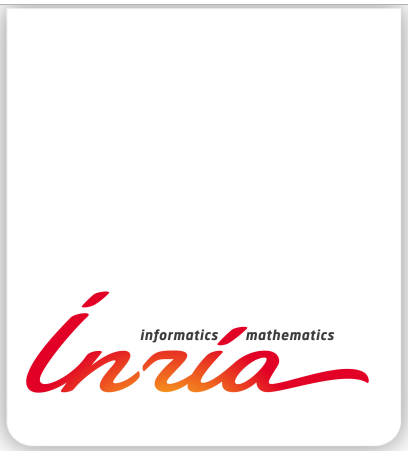
[23] M. Shapiro, N. Preguiça, C. Baquero, et al. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pp. 386–400, Grenoble, France, Oct. 2011.

[24] M. Shapiro, M. Saeida Ardekani, and G. Petri. Consistency in 3D. In *Int. Conf. on Concurrency Theory (CONCUR)*, volume 59 of *Leibniz Int. Proc. in Informatics (LIPICS)*, pp. 3:1–3:14, Québec, Québec, Canada, Aug. 2016.

[25] The SyncFree Consortium. AntidoteDB: A planet-scale, available, transactional database with strong semantics. Website http://antidoteDB.eu/.

[26] G. Tomás, P. Zeller, V. Balegas, et al. FMKe: a real-world benchmark for key-value data stores. In *W. on Principles and Practice of Consistency for Distr. Data (PaPoC)*, Belgrade, Serbia, Apr. 2017.

[27] M. Zawirski, N. Preguiça, S. Duarte, et al. Write fast, read in the past: Causal consistency for client-side applications. In *Int. Conf. on Middleware (MIDDLEWARE)*, pp. 75–87, Vancouver, BC, Canada, Dec. 2015.



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399